1-1-2004

Software tamper resistance through dynamic monitoring

Brian Blietz
Iowa State University

Software tamper resistance through dynamic monitoring

by

Brian Blietz

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Akhilesh Tyagi, Major Professor
Soma Chaudhuri
Doug Jacobson

Iowa State University

Ames, Iowa

2004

Graduate College
Iowa State University

This is to certify that the master's thesis of

Brian Blietz

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Today's complex software is of much value to its creator. Whether that be a company with many products, or the only product of a small company. Of much concern is the protection of this software, such that it will always retain the functionality which its creators intended, always protect the intellectual property embedded in the program, and thwart attempts to make illegal copies of the program. There have been several different approaches proposed in an effort to deal with such attacks, but most of the commercially available defenses rely on reactive measures. Almost anyone will agree that software should be protected, but little is agreed upon as to how this should be done. At the root of the problem is the need for a solution that relies on proactive measures, which ultimately means modifications to the way software is made. Furthermore, a solution that requires no new software design paradigms (from a software engineer's viewpoint) and is fully automated is highly desirable.

1.1 Proactive Solutions

In order to take a proactive approach, one must be able to either determine that the program has been altered, or make any alterations impossible. Tamper-proofing and code obfuscation attempt to address these requirements. What is of much interest, as we will see later, is that these two approaches can be combined for a robust and tamper-resistant solution.

In order for a program to detect whether or not it has been altered requires that the program check itself every time it is run. Because it must be checked at runtime, the most obvious solution is to insert tamper-proofing code into the program itself.

The other approach is to render a program unintelligible to its adversary through obfuscation. The basic premise being, that if you don't know what you are looking at, then it is impossible

to intelligently alter the code. One problem that arises from this approach is that it is possible to obfuscate a program to such a degree, that even the creator can no longer tell what he/she is looking at. This can cause major problems when one must debug software through the use of stack dumps, assembly traces, and/or memory inspection. One would argue that the solution to this problem would be to debug the code prior to obfuscation, but this may not be possible. After a piece of software is obfuscated and deployed, the end user may experience bugs that were not seen during testing of the original code. Tracking down problems at this stage may become impossible.

A tamper resistant approach that detects and/or subverts/corrects the tampering actions in real time (concurrently with the program execution) is desirable. Ultimately, a technique that will protect the software transparently, without the user even knowing such actions are taking place, will succeed.

1.2 What We Are Up Against

Observation of the historical trends suggest that the attack methods appear to be more mature than (lead in time) the security methods. Attacks use many readily available tools which allow them to monitor network connections, monitor a program's instructions with debuggers, modify an operating system's kernel[1], monitor address and memory busses, etc.. It seems somewhat ironic that the tools used to help design and implement complex software, are the same tools used to attack it.

Much has been done to thwart network originated attacks, but little has been done to thwart hardware and software based attacks on the intellectual property embedded within a program. These attacks include modifications to a program to skip crucial checks (such as license file/servers), or reverse engineering of a key piece of a program's functionality.

The anti-tamper techniques in general are designed to detect or sense any type of tampering of a program. Once such tampering is detected, one of many possible actions could be taken by the anti-tamper part of the software. These actions could include disabling the software,

[1]Operating systems such as Linux are open source, allowing modification of any part of a program's system interface.

deleting the software, or making the software generate invalid results rendering it useless to the tampering adversary.

1.3 Control Flow Monitoring Using a Two Process Model

This thesis proposes one such anti-tamper methodology based on constant program monitoring. The monitoring process must have some knowledge of the monitored program's meta-structure and some notion of program semantics with respect to the tamper protected domain. This domain happens to be the control flow integrity. We propose that all tampering methods, be they data tampering, memory tampering or network traffic tampering, eventually exhibit themselves in the control flow corruption. Hence, all types of tampering can be captured even if only control flow tampering is incorporated into an anti-tamper system design paradigm.

With this premise, we propose to use a two-process model for program integrity (one that is not tampered) checking. The original program runs as a program process (P-Process), whereas a monitoring co-process (M-process) runs concurrently with the sole objective of dynamically verifying the control flow of the P-process.

The original program is compiled into the two process model by a modified Gnu C Compiler (*gcc*). The P-process performs periodic control flow integrity checks by communicating its instantiated control flow (since the last check) to the M-process. The M-process has the correct control flow of the P-process stored in it. The compiler can statically determine the piecemeal control flow information and compile it into a data structure resident in the M-process memory.

The M-process performs the integrity check on the received control flow segment with respect to this data structure. If the check fails, it can take one of the few corrective actions such as killing the P-process and/or raising an interrupt. If the check passes, no information need be communicated back to the P-process. The frequency of the *check* primitives is user specifiable to control the overhead of the scheme.

We have implemented this scheme with *gcc* on Linux. We present the performance overhead data on a variety of programs (interactive versus CPU intensive) in Chapter 4. The implementation details are given in Chapter 3.4.

It is worth noting that the entire monitoring framework was first conceived as an architectural paradigm. The processor in such a case would have two computing engines: one for the P-process and one for the M-process. Such a two-instruction stream processor can perform these checks much more efficiently and stealthily.

We will show how this approach pro-actively protects a program with no interaction (and no knowledge required) from the end user. Furthermore, we will show how this approach is highly suitable for user-interaction dominated programs, and configurable for computationally intensive programs.

CHAPTER 2. BACKGROUND AND RELATED WORK

There exists a wide range of tamper resistance methodologies. The following discusses some of the more widely know approaches. The main focus of this paper is on control flow monitoring, augmented with some new approaches, which are discussed in subsequent chapters.

It is also important to keep in mind, that in order to increase the effectiveness of tamper resistance, multiple approaches can be combined. One should think carefully about how to combine different approaches, and strive to mask the weaknesses of one, with the strengths of another. For example, combining control flow monitoring with obfuscation can lead to a monitored program that requires significant effort to reverse engineer (NP-Complete)[12].

2.1 Watermarking

Watermarking consists of statically, or dynamically inserting signatures into a program, which serve to identify the original owner. Static watermarks never change, and are therefore subject to some level of reverse engineering. Dynamic watermarks change with the program execution. Watermarks are either extracted from a program's image, or from the program execution itself. Watermarking, as mention previously, is a reactive measure. Hence, we will not be looking into watermarking as an effective technique. While this performs a valuable function, the idea is to avoid the need for this all together by making the program impossible to tamper with in the first place. Good representatives of software watermarking methods are [2] and [11].

Static watermarks evolved from the area of digital imaging. Watermarks for images have been around for quite some time, and are fairly mature. This idea has been used in program watermarking in a very simple way. First watermark a small image. Then embed the image into the data section of a program. This way, the image can be extracted from the program,

and then the watermark from the image. It is rather apparent that such a simplistic approach is easily broken using standard binary editing tools.

Of more interest are the dynamic water marks. For example, and Easter Egg watermark is a watermark that is embedded into the functionality of the program. When the program is given a particular input set, it performs some action that is immediately visible to the user. A typical Easter Egg watermark might display a logo, or force a program into a particular mode. For example, the following will turn Microsoft Word97 into a pinball game:

1. Open a new document
2. Type the word "Blue"
3. Highlight the word "Blue"
4. Using the `Format` menu select `Font`
5. Choose `Font Style` Bold, `Color` Blue
6. Type " " (space) after word "Blue"
7. Using the `Help` menu select `About`
8. `Ctrl-Shift-Left click` the Word icon/banner
9. Use Z for left flipper, M for right flipper, and ESC to exit

Other dynamic watermarks include execution tracing and data structure analysis, both producing no immediate output for the user, but instead relying on monitoring a particular property of the program when given special input. Because of the nature of these two watermarks, they do not work well with most types of code obfuscation.

2.2 Obfuscation

Code obfuscation attempts to make the task of reverse engineering a program daunting and time consuming. This is done by transforming the original program into an equivalent program, which is much harder to understand, using static analysis.

More formally, code obfuscation involves transforming the original program P into a new program P' with the same black box functionality. P' should be built such that [2]:

i It maximizes obscurity, i.e., it is far more time consuming to reverse engineer P' when compared to P.

ii It maximizes resilience, i.e., P' is resilient to automated attacks. Either they will not work at all, or they will be so time consuming that they will not be practical.

iii It maximizes stealth properties, i.e., P' should exhibit similar statistical properties, when compared to P.

iv It minimizes cost, i.e., the performance degradation caused by adding obfuscation techniques to P' should be minimized.

Obfuscation techniques involve lexical, control and data transformations. Lexical transformations alter the actual source code, such as Java code. This transforms the original source code into a lexically equivalent form by mangling names and scrambling identifiers. Such transformations make it a daunting task to reverse engineer a program. A simple example would be to swap the names of the functions `add()` and `subtract()`. (This would also involve swapping every reference to these functions as well.) An even more interesting approach would to replace `add()` with the function `tcartbus()` and replace `subtract()` with the function `sunim()`.

Control transformations alter the control flow of the program by changing branch targets to an ambiguous state. The code for the program is shuffled such that the original branch targets are no longer correct. During this shuffling, the new targets are calculated, and code is inserted in place of the old branch instruction to acquire its new target address.

Data transformations rearrange data structures such that they are not contiguous. Data can be transformed all the way down to the bit level. Bit interleaving is one example.

One particular obfuscation technique of interest is obscuring control flow of a program. By obscuring branch target addresses, static analysis of a control flow graph can be shown to be NP-hard [12]. Program address based obfuscation is presented in [7].

2.3 Hashing functions

Hashing functions scan sections of the program, and use the data contained therein as input to a mathematical equation. The simplest of which would be to sum all the numbers in a given section. When done, the answer must agree with the previously determined result. If they are not the same, this section of the program has been altered.

It is fairly obvious that such a scheme is easily broken, which has led to more complex techniques. Some techniques may use values on the stack at a crucial instant in time, or values in a register. Others perform complex mathematical equations on overlapping sections of code. Horne et al. [5] have implemented such a system, using linear hash functions, which overlap and also hash the hashing functions as well. One of the strong points of hashing is that you can have as many hashing functions as you want, all performing a different type of hash.

2.4 Control Flow Monitoring

Control flow monitoring involves tracing the execution of a program as it is running. A program is broken down into basic blocks using a Control Flow Graph (CFG) representation. Code is inserted into each of the basic blocks in order to keep a trace of the running application. At certain intervals, the trace is checked against a known good trace which is determined at compile time. This particular approach will be covered in much more detail in the remaining chapters of this thesis.

2.5 Types of Attacks

When one decides to prevent attacks on their software, they must first decide what type of attack is of most concern to them. The following shows attacks classified into three basic categories. The main distinction being that each of these depend on the relative location of the origination of the attack.

i **Outside attackers** attempting to gain entry over a networked connection. This is the most common type of attack today, and several preventive measures are already

in place.

ii **Executable code** that is run on a target system, but not under the direct control of the attacker, such as viruses and Trojan horses. This is a fairly common attack which has several preventive measures already in place as well.

iii **God Mode attacks:** The attacker owns a copy of the software, and has complete control over the system it is run on. This is one of most damaging attacks in that it allows the theft of Intellectual Property, and the execution of pirated software.

The God Mode attack model assumes that the attacker has full control over the system, *i.e.*, the attacker owns the system the program is running on, and has total access to the software and hardware in the system. The attacker may choose to run binary analysis tools, software and hardware debuggers, logic analyzers, etc.. The main hurdle for the attacker is rooted in the amount of technical know-how he/she possess. These type of attacks are the focus of this paper.

2.6 Control Flow Graph Overview

A control flow graph (CFG) is an abstract data structure, which encompasses the procedural flow of a program. Each node in a CFG corresponds to a basic block, which is a maximal sequence of instructions with exactly one entry and one exit. A jump target constitutes the beginning of a basic block, while a jump instruction signifies the end of a block.

There are two special basic blocks that every program must have, the *entry block* and the *exit block*. The entry block is the location where control enters the program, and the exit block it where control exits the program.

The CFG is a static representation of a program, and covers the program in its entirety. Compilers use this data structure for several optimization passes, one such example being detecting dead code. Dead code is easily detected by examining the entries into basic blocks. If a block has no entry, then it will never be executed.

Each basic block has at most two successors, and at least one predecessor (except for the entry block for a program which has no predecessor). At a high level, basic blocks are most

commonly created from conditional statements such as *if then else* statements. The *if* block is the first successor ,and the *else* block is the second successor.

Figure 2.1 shows a simple example of how an *if then else* statement is decomposed into basic blocks. One must keep in mind though, that the CFG a compiler maintains does not refer to high level code, instead it more closely resembles low level code, such as assembly.

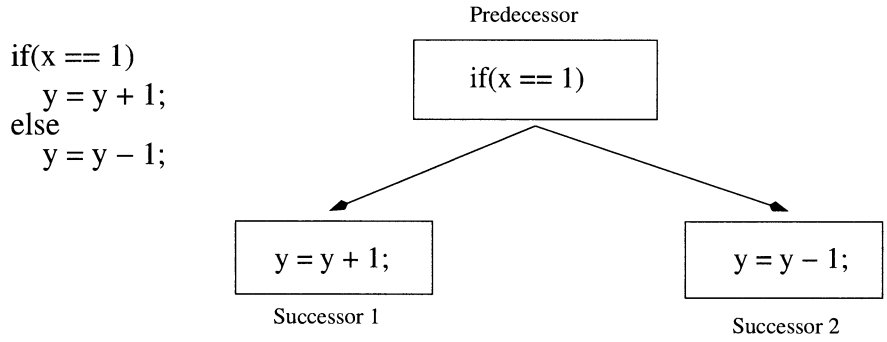

Figure 2.1 Example C source code and its corresponding CFG

Figure 2.2 shows how the previous *if then else* statement is decomposed into its corresponding assembly.

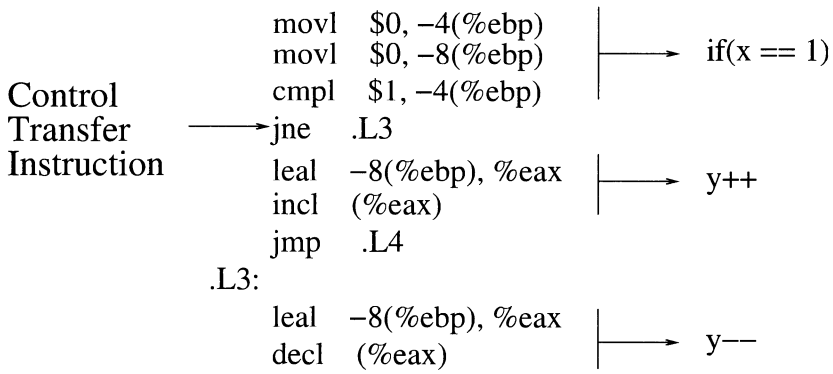

Figure 2.2 Example assembly code

When the source code is compiled, the compiler translates the source into an intermediary representation called the Register Transfer Language (RTL). GCC maintains a representation

of the CFG for a program based on its RTL. It would be of little use to have the CFG refer to C level source code, due to its higher level of abstraction. RTL code is very similar to assembly code. In fact, the RTL is used directly to generate the assembly code, which is then used by the assembler to generate object or executable files.

When the source code is in the RTL representation, the CFG is readily available to the compiler.[1] The CFG is used by several other optimization passes internal to GCC, and is updated accordingly after each pass. When GCC is ready to output the assembly code for the program, the CFG is in its final state. This is where the tamper resistant code is inserted.

By exploiting the fact that the CFG represents every possible path within a program, one can detect tampering of the software, with a high degree of certainty. By monitoring the control flow of a program, one can determine if it has been altered in any way, by simply comparing it to the CFG that was obtained when the program was compiled. If at any time, a path is taken that is not accounted for in the CFG, the program had to be altered in order to do so. This type of alteration could be accomplished by using one of many different binary editing tools available.

Take for example, a single function that checks to see if a license file is present, and valid. This function will have an entry and exit point associated with it. It is conceivable that an adversary could edit the instructions (at the binary level) of the entry block, such that it simply returns a positive outcome, such as a successful return code. Thus skipping the entire check all together!

By monitoring the control flow of the program, such an occurrence could be detected by evaluating the flow to the CFG. In the previous example, the license check could be part of the program's initialization function. All of the possible paths through this function are contained in the CFG. By checking all the possible paths that lead up to, and away from the license check section, one can check if the program was altered in some way to skip over the check.

Facilitating such a process involves inserting a separate instruction stream, which performs the monitoring of the program. Our architecture calls for a two process model in which the

[1]GCC has incorporated a CFG library into versions greater then gcc3.0. Other compilers have similar support such as SUIF2 from Stanford.

application (P-Process) communicates its control flow to a monitoring process (M-Process). In order to enable the P-Process to communicate with the M-Process, code is inserted at the beginning of each basic block. This code's purpose is to inform the M-Process of the unique ID assigned to this block. The M-Process will gather the trace information for basic blocks, and check them against the known CFG representation, that was determined at compile time. When it finds a control flow sequence that is invalid, it knows that the M-Process has been tampered with in some way. At this point, appropriate actions can be taken to secure the application from an attacker. Such actions can range from printing a warning message, to completely deleting the file from the disk. The action taken is application specific, which allows different *penalties* for tampering with a given program.

The CFG obtained from the compiler is a static representation of the flow of the program. Static analysis of the CFG is of much concern due to an attack's ability to automate a reverse engineering attack on the control flow. It is conceivable that someone could disassemble the program and recreate the CFG from the assembly code. This would allow an attack to succeed by communicating the fake set of unique identifiers associated with the original control flow graph instead of the new basic blocks inserted. This will allow the tampering to go undetected.

In order to prevent such an attack, the unique identifiers of each basic block must be dynamically determined at run-time uniquely for each run. Before the M-Process starts the P-Process, it will change the unique identifier associated with each basic block by replacing the value used by each basic block in its communication phase. The M-Process will also update its local CFG table with the new value.

To further protect the CFG table, the M-Process will remain in a mostly encrypted form during its execution. The M-Process will use Integrity Verification Kernels (IVK) variant [1] to allow only a small portion of the M-Process to be in plain text at any given time. This protects the CFG data structure from static analysis. As pointed out previously, static analysis will only give an abstract representation of the CFG. The actual unique identifiers that make up the CFG will be different with each run.

2.7 Monitor Process Overview

The M-Process is an application specific process that comes paired with the actual executable for the P-Process. Its main responsibility is to monitor the flow of the P-Process, and detect if any tampering has been performed. Consequently, such a process should be stealthy, compact, and efficient.

In addition to monitoring the P-Process, the M-Process is responsible for a myriad of other tasks such as starting the P-Process. When the M-Process starts up, it first performs some fix up tasks on the P-Process[2]. It then *execs*[3] the P-Process which allows it to start.

Once the P-Process is running, for every basic block in the P-Process, a unique identifier will be sent to the M-Process. The M-Process will then verify the correct execution of the program by comparing it to the CFG that was determined at the compile time. The CFG data structure can be seen as a simple array containing 3-tuples of `(Parent, Child1, Child2)` entries for the entire CFG.

The M-Process will initially be in an encrypted form, using standard encryption methods. An IVK is decrypted, and allowed to execute, one at a time. When any portion of the executable code or the data section is needed, it is modified from its encrypted form to its plain text, decrypted form, which is then executed or read as data [1].

Having the M-Process stored as an encrypted file buys a significant advantage over a plain text M-process scheme. Not only will the CFG data structure be protected from static analysis, but several parts of the P-Process will reside as data structures in the M-Process as well.[4] This is discussed in further detail in Chapter 3.4.5 and Chapter3.4.6.

[2]The P-Process is not in a run-able state until the M-Process restores several parts of its executable code.

[3]exec refers to anything similar to the execv() call in Linux/Unix.

[4]Before the P-Process is paired to the M-Process, several parts of the P-Process are stripped out, and corresponding data structures are inserted into the M-Process. For example, the entire `main()` routine.

CHAPTER 3. A CONTROL FLOW MONITORING IMPLEMENTATION

3.1 Application Transformation

The application, or P-Process, must be transformed such that a new P-Process, P' is created with the same black box functionality as the original process P. This means that any changes made to the P-Process must be transparent to the user. The main goal is to create a tamper resistant version of P that is protected without any interaction from or inconvenience to the user.

3.1.1 Instruction Insertion

Instructions must be inserted into each basic block in order to communicate trace information to the M-Process. For example, consider a simple *if then else* statement as shown in Figure 3.1.

```
if(x == 1)
   y = y + 1;
else
   y = y - 1;
```

Figure 3.1 Example C source code

First decompose it into its basic blocks at the assembly level.[1] Figure 3.2 shows BB0, BB1, and BB2 which refer to the three basic blocks that comprise the simple *if* statement in Figure 3.1. Notice the labels that start with the prefix ATP that correspond to the beginning of each basic block. The number in each label corresponds to its unique identifier.

[1] All assembly given uses the x86 instruction set. All assembly is created using GCC3.2.3.

Next these labels are replaced with instructions to communicate the block's identifier to the M-Process. Figure 3.3 shows the inserted instructions for each of the steps outlined below.

1. Setup the stack pointer.
2. Push the unique identifier onto the call stack.
3. Call a function to write to the unique identifier to the pipe that the M-Process is listening on.[2]
4. Restore the stack pointer for the current function.

3.1.2 Communication

At this point, each basic block in the P-Process has the extra instructions to communicate its basic block identifier to the M-Process. In a perfect world, this would be all that is needed, but such an implementation is not feasible. In order to implement this functionality, the P-Process must also tell the M-Process when it should verify the current trace that it is keeping. This can be done by transmitting a *magic* number, i.e. the VERIFY code.

The key to making this work, is in determining where to insert the VERIFY code transmission to the M-Process. Inserting the VERIFY too often will yield very short traces, which will expose VERIFY boundaries in which an adversary would insert code. Limiting these boundaries will have a significant effect on how robust our mechanism is. In the worst case, a VERIFY could be inserted into every basic block, thus nullifying the trace mechanism all together! Also, a short period between the verification also leads to large overhead.

On the other hand, due to compiler limitations, function boundaries pose a problem for long trace lengths. The CFG library in GCC does not allow basic blocks to be broken based on a function call. This is understandable because, in the worst case, the function call could be in a library. If the function is in a library, then it is impossible to have a usable successor for the block. On top of this, because of the way GCC works, it is also very complicated to have a

[2] A separate source file was used to implement the P-Process communication. This file is compiled to an object file, and linked with the P-Process.

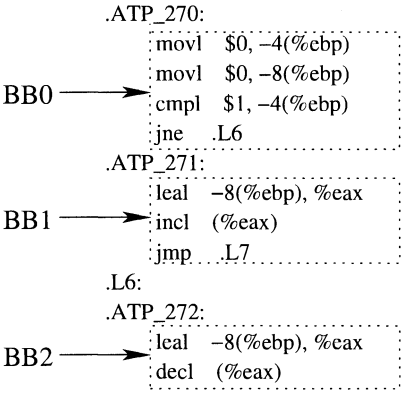

Figure 3.2 Assembly output with ATP labels

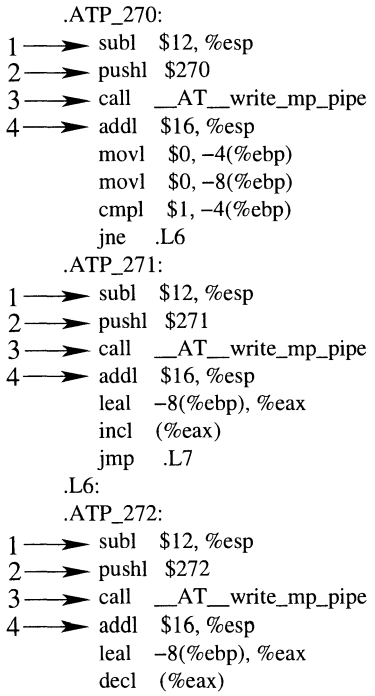

Figure 3.3 Modified assembly code with communication instructions inserted

successor that is in another file or even in another function. GCC processes one function at a time, when it performs its passes. In other words, the CFG is only maintained per function, and not globally.

From an implementation standpoint, there are several levels of abstraction that one may decide to insert the VERIFY codes. Inserting the VERIFY at the RTL level has several advantages, such as portability. It also lends itself to allowing the compiler to generate efficient assembly code. The main disadvantage to this approach is the complexity of its implementation. A much easier approach is to insert instructions directly into the assembly code for a program. The assembly level approach is the one that we are using currently.

Now, we need to decide when and where to insert the VERIFY codes. It is fairly easy to place the VERIFY code transmission at the end of every function. Through the use of temporary files, global CFG information is be maintained for each function's entry block identifier. At the assembly level, this is done by parsing the assembly instructions to determine the callee for each function call. Once a *call* instruction is seen, the global CFG data file is scanned to determine the entry block for the callee, and the successors for the current block (in which the call is contained) are updated to point to the entry block of the called function.

3.2 Monitor Process

The Monitor Process (M-Process) resides along side the P-Process. It's purpose is to monitor the control flow of the P-Process, and take appropriate actions when an inconsistency is found. The M-Process structure is shown in Figure 3.4. The following sections describe each of the components for the M-Process.

3.2.1 Communication

In order for the M-Process to receive identifiers from the P-Process, it must have a communication port of some sort. At the software level, this is implemented as a Unix pipe, in which the M-Process listens on. The M-Process will sit and wait for any identifiers that are sent from the P-Process. As the identifiers are received, they are collected in a trace buffer for further

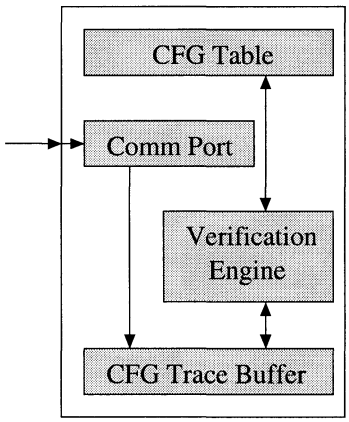

Figure 3.4 M-Process

use by the verification engine. Typically, the M-Process will wait for a VERIFY code from the P-Process before it invokes the verification engine. But, due to a fixed trace buffer size, a full buffer will force a check as well. A common situation that would fill up the buffer before a VERIFY code arrives, is a *for* loop that iterates many times.

Figure 3.5 shows the communication from the P-Process to the M-Process. This uses the simple *if then else* statement shown in Figure 3.1. The path that is shown is the flow that would be seen if the *if* statement evaluated to true (1).

3.2.2 Control Flow Graph Representation

Let V be the set of nodes, and E be the set of edges in $V \times V$ that connect the nodes in V. Let C be the CFG of a program such that the set $C = \{V, E\}$, where $V = \{v_1, v_2, ...v_k\}$. Each v_i is a node of the CFG, and each $e_{i,j}$ is an edge that represents a control transfer from v_i to v_j.

Next define $succ(v_i)$ to be the set of nodes that are successors of v_i, and $pred(v_i)$ to be the set of nodes that are predecessors of v_i. A node v_i belongs to succ(v_j) if and only if $e_{i,j}$ exists. Similarly, a node v_j belongs to $pred(v_i)$ if and only if $e_{j,i}$ exists [4].

As the P-Process is compiled, a CFG is created. This CFG consists of 3-tuples in the form of {P_i,$S1_i$,$S2_i$}, where P_i is the current basic block's identifier, and $S1_i$ and $S2_i$ are the two successors for this block. Each node (v_i) has a corresponding P_i. For every P_i there are at

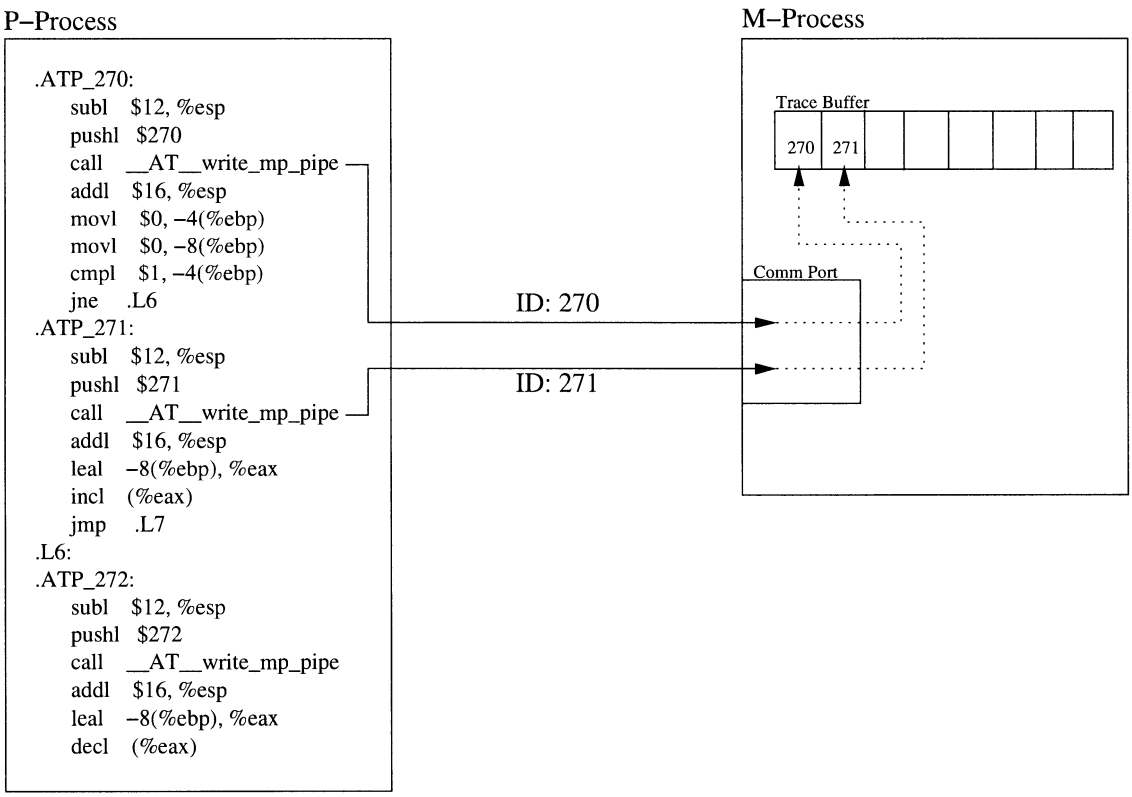

Figure 3.5 P-Process to M-Process communication example.

most two successor edges. Each e_i corresponds to an edge from node P_i to one if its successors. By evaluating the edges, $S1_i$ and $S2_i$ are calculated. This can be done simply by using $e_{i,1}$ to determine $S1_i$ and $e_{i,2}$ to determine $S2_i$.

This information is kept in a linear array, which can be searched based on these 3-tuples. This is the simplistic description. Many enhancements such as sorting, caching, and dynamic identifiers help serve to increase performance further, and to better secure the CFG data structure.

3.2.3 Verification

Conceptually, during execution of the program, if any $e_{i,j}$ exist that are not part of C, then an illegal control flow has been seen. Verification of a trace of identifiers involves sequentially

scanning the trace buffer and comparing it to the CFG. To do this, let P_i equal the current identifier in the trace buffer, and let P_{i+1} be the next identifier in the trace. Scan the CFG table until a tuple containing P_i as $\{P_i, S1_i, S2_i\}$ is found. The following two entries $S1_i$ and $S2_i$ are the two successors of this basic block in a legitimate control flow. Each entry P_i is tested such that P_{i+1} must equal either S_i1 or S_i2. If this test fails, an incorrect path has been taken in the program, indicating that the program has been tampered with.

Figure 3.6 shows some simple pseudo code used to verify the trace buffer. It should be noted, that the last entry in the trace buffer is never used for a parent identifier. Its only purpose is to be used as a successor identifier. This is because it is the last entry, and subsequently has no successor in the trace buffer to test.

```
for i = 0 to number of entries in trace–buffer – 1
   let P = trace_buffer[i]
   search CFG table for tuple {P,S1, S2} using P as key
   let succ1 = S1 and succ2 = S2
   test: trace_buffer[i + 1] == S1? OR trace_buffer[i + 1] == S2?
   if test fails, GOTO FAIL
continue
```

Figure 3.6 Verification pseudo code.

3.2.4 Protection Scheme (Aucsmith's IVKs)

Aucsmith[1] proposed a very interesting tamper resistant scheme. This scheme however has very high overhead. A simplified version of this scheme is presented in the following. For a detailed treatment of an implementation of this scheme, the reader is referred to [3].

Each text section is broken down into several text subsections. The objective is that at any point in time, exactly one text subsection is in plaintext (the one that is currently executing). Whenever, there is a control flow edge between two such text subsections, the target text subsection is decrypted into plaintext alongwith encryption of the source text subsection simultaneously. Figure 3.8 illustrates a program (text section) broken into 8 text sub sections

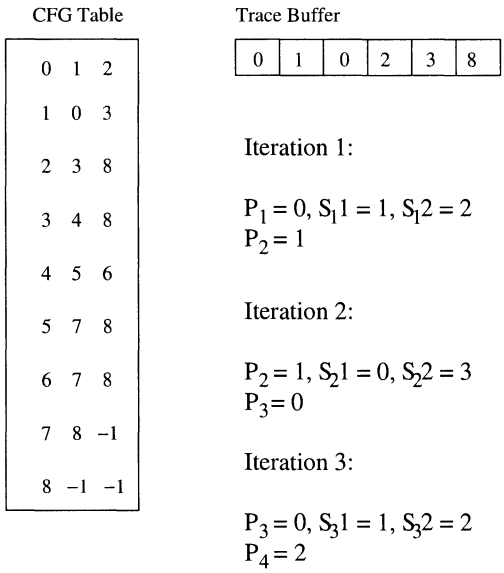

Figure 3.7 M-Process Verification Example

and the induced control flow graph at the text subsection granularity. Note that there can be multiple control flow edges between two subsections as in T_6 and T_7. There can be backward control flow edges as from T_5 to T_2.

Initially, all the text subsections T_1 through T_7 are in memory in some encrypted form (to be described later). Only the entry subsection T_0 is in plaintext. Let us say that the first inter-text-subsection control flow edge to be instantiated is from T_0 to T_3. The following actions are taken at that point.

Transfer control to a "decrypt & jump" module (similar to Aucsmith's) with a key as an argument. The key for the control flow edge from T_0 to T_3 is denoted by $K_{0,3}$. The decrypt & jump action XORs the key $K_{0,3}$ with each subsection. If the text subsections were assigned appropriate initial states, exactly one of the subsections would appear in plaintext. In this case, the text subsection T_3 must have been initialized to $T_3 \oplus K_{0,3}$, which would result in T_3 decrypting into plaintext from XOR with $K_{0,3}$. Each text subsection can either have a magic number or a special nop instruction (or some null instruction such as jump to next location) embedded at the beginning. The decrypt & jump function can check for that special instruction

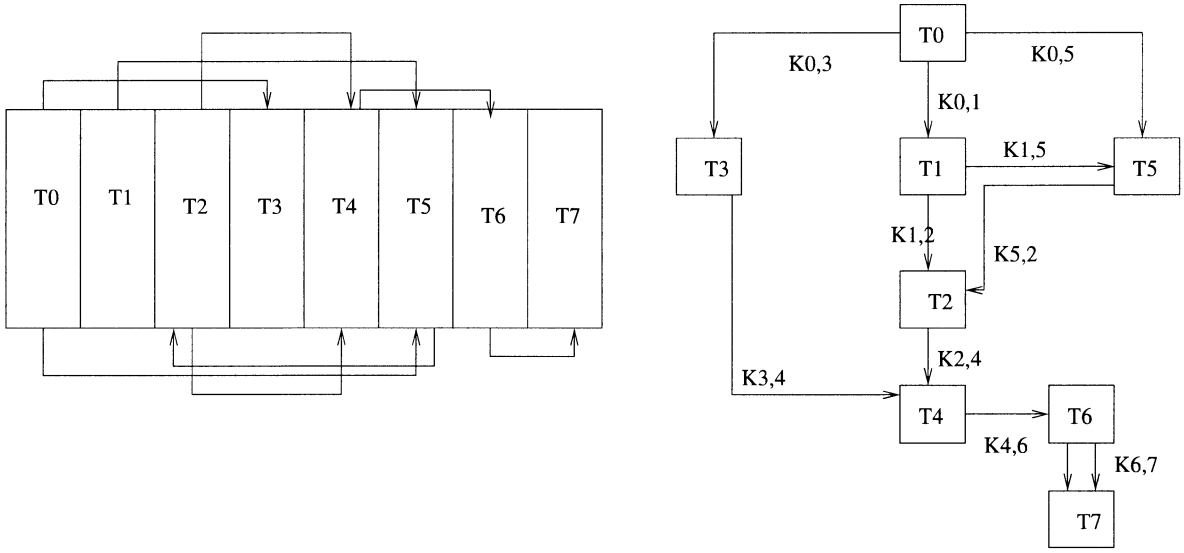

Figure 3.8 Text Subsection Layout

or magic number at the beginning of each text subsection. The text subsection with the special entry attribute is the one the control is transferred to. Note that a branch/jump instruction from T_0 only needed to specify the key $K_{0,3}$ to specify its target (and an offset from the beginning of the target text subsection T_3 to the actual branch target).

The assumption is that each text subsection is designed to be the same size (it need not be as it depends on the key size). Maximum obfuscation is obtained if the key is as large as the text subsections (TS bytes). However, then the keys are large. The keys could be chosen in other granularities as well such as word size (4B) or cache block size (say 16B) or anything else up to TS. This is a tradeoff in memory overhead to maintain the keys versus obfuscation. If the key is chosen to be cache block size, then each cache block sized chunk in a text subsection is XORed with the key.

Key Consistency: Consider the text subsection T_4. It could either be reached from T_0 through T_3 (path $T_0T_3T_4$) or with path $T_0T_1T_2T_4$. How should we initially encrypt the subsection T_4? If we make it consistent with the path $T_0T_3T_4$ then the initial state ought to be $T_4 \oplus K_{0,3} \oplus K_{3,4}$. Then the control flow edge from T_0 to T_3 would have XORed the initial $T_4 \oplus K_{0,3} \oplus K_{3,4}$ with $K_{0,3}$, and the control flow edge from T_3 to T_4 would have XORed the

resulting $T_4 \oplus K_{3,4}$ with $K_{3,4}$ exposing T_4 as desired. But the path $T_0T_1T_2T_4$ requires an initial encoding of $T_4 \oplus K_{0,1} \oplus K_{1,2} \oplus K_{2,4}$. Which one should it be? In general, many more paths could have led from the root node to this node placing many more constraints on the initial encoding. Can we always find consistent set of keys to satisfy all these constraints?

It turns out that the only constraint the keys need to satisfy is that if keys K_{i_1}, K_{i_2}, ..., K_{i_k} label the control flow edges along a cycle (undirected induced) then $K_{i_1} \oplus K_{i_2} \oplus \ldots \oplus K_{i_k}$ must be 0! Note that $\oplus$ signifies bit-wise XOR of its arguments. By this token, in the example above, $K_{0,3} \oplus K_{3,4} \oplus K_{2,4} \oplus K_{1,2} \oplus K_{0,1} = 0$ since $T_0, T_3, T_4, T_2, T_1, T_0$ form a cycle. This would imply that $K_{0,3} \oplus K_{3,4} = K_{2,4} \oplus K_{1,2} \oplus K_{0,1}$ and hence these keys can be assigned consistently.

The general strategy would be to choose all the keys but one in a cycle randomly independently. The one key would have to be derived from all the other keys through the cycle constraint.

Text Subsection Partitioning: One of the objectives of the text section partitioning into text subsections could be to minimize the number of cross-subsection control flow edges. This is since each instantiated control flow edge costs a decrypt & jump operation. Hence, a k-mincut of the control flow graph of a text section into k text subsections of more or less equal size (with a constraint on the upper bound on this size) is the optimization objective. If profiling information is also available annotating the control flow edges with the probability of instantiation, the mincut gives us an even better partition. An approach based on Kernighan-Lin mincut heuristic [8] will provide a reasonable cell partitioning.

3.3 Utility Applications

After *gcc* has processed the source code, and inserted the corresponding labels, several utilities are used to insert the anti-tamper instructions directly into the assembly output. The following sections will describe each of the utilities, and detail the service each performs.

3.3.1 Function Boundaries

When the source is compiled, *gcc* maintains nothing with respect to function call boundaries. Technically speaking, a function call constitutes a control transfer, but the CFG library provided with *gcc* does not take this into account. This causes problems when performing predicate verification.

In order to correctly verify predicates, as they cross function boundaries, **fixpredtable** is used to parse the assembly, detecting function calls. When a function call is seen inside a basic block, the successor for the basic block is updated to contain the unique identifier of the entry block to the function that is being called.

It is worth noting that this is an advanced option of the anti-tamper suite, and need not be used, unless so desired. If such a mechanism is not needed, one only needs to simply place a VERIFY code transmission at the beginning of each function.

3.3.2 Instruction Insertion

The assembly output from *gcc* will contain labels signifying the beginning of each basic block. These labels determine where to insert the communication instruction sequence for each basic block. An example can be seen in Figure 3.3. **addmpcalls** will parse the assembly output and add the correct instruction sequence after each corresponding **ATP** label.

3.3.3 Dynamic Branch Targets

Certain programming scenarios require the use of *callback* functions. Such functions are usually passed to another function as a pointer to a function, such that the function may ambiguously call a function to perform a task. A classic example is found when processing *widgets* in a graphical user interface environment. One can create a button and associate a callback function with the button, which will perform the appropriate task when the button is pressed.

Such functions pose a problem when determining trace information for a program. Because the callback function is assigned dynamically, it is impossible to determine at compile time which

function will actually be called with 100% accuracy.

The utility **fixbadjumps** searches the assembly code for such instances of jumps to callback functions. When it sees such an event take place, it inserts a VERIFY code communication sequence before the jump instruction. This allows the M-Process to validate the current trace, and proceed normally when control enters the callback function.

3.3.4 Instruction Removal

The *main* function of the P-Process is stripped out, rendering the application unusable in its raw state. **stripappmain** performs this task by using the symbol table to locate the main function. Once it finds main, it replaces it with all zeros, and creates a data structure to be placed into the M-Process.

3.3.5 CFG Template Insertion

When the P-Process is compiled, a temporary data file is created, which contains the CFG. This CFG is merely a template which is used to map the dynamic identifiers to their original ones. **addpredtable** will insert this template into the M-Process as a data structure, which the M-Process will use at runtime.

3.3.6 Instruction Hiding

addbinimg will strip out several key parts of the P-Process, and put them into the M-Process as data structures. This is discussed in detail in Section 3.4.5 and Section 3.4.6.

3.3.7 Runtime CFG Table Storage

In order for the M-Process to dynamically assign unique identifiers a place holder is used for this data structure. It is felt that this is more secure than simply using *malloc* because this data structure can be protected using encrypting methods via Aucsmith's scheme. If memory is simply allocated using *malloc* or some similar memory allocation function, the real CFG would

be visible in memory to the adversary. **addcfgtable** will create an array, with all zeros, the size of the CFG table, which will be filled in at runtime.

3.3.8 File Size Check

In order to combat attacks that may add code to the P-Process, the file size is recorded. **addfilesize** will place the correct file size into the M-Process, so the M-Process can check the actual file size against the compiled file size. This is a very simple, yet effective, countermeasure which will keep the adversary from inserting fix-up tasks to fool the M-Process.

3.4 Implementation

Conceptually, CFG monitoring is a fairly straight forward idea. Simply analyze a program at compile time, save that information, and compare it against the runtime flow of the program. But, from an implementation standpoint, things are not quite as easy. Issues such as performance, static analysis, and code stability all lend to a complicated implementation. The following sections will discuss these issues in more detail.

3.4.1 CFG Performance Enhancements

The CFG data structure is typically a large entity. Basics blocks are usually quite small, on average 10 instructions or so. This means that searching the CFG can become a costly task to perform. In the worst case, the identifier could be the last entry in the table, therefore requiring the search to look at every entry in the table. For large tables, a significant performance hit was observed.

One method to help speed up the CFG table search is to implement a caching mechanism. The cache would contain the last N 3-tuples which correspond to the last N identifiers seen, where N is the number of entries the cache can hold. The cache is implemented as a simple FIFO buffer, which holds a trace window for the most recent identifiers that the M-Process has seen. The M-Process will first check the cache to see if an entry is already there. Only if it is not found, will it perform a search on the full CFG. Due to locality of reference inherent in software

programs, this improvement increases performance greatly. For ease of implementation, and more notably speed, duplicate entries in the cache are allowed.

Another method to speed up this operation is to sort the CFG table based on increasing P values, and perform a binary search. This method yields a search time of $O(\log N)$, where N is the number of basic blocks. Because this yields a vast improvement, the cache size can be reduced considerably. Earlier, when there were thousands of entries in the CFG table, a fairly large cache yielded good performance gains. However, with binary search, cache sizes larger than 32 entries actually yielded poorer performance.

3.4.2 System Library Flaws

A possible security hole is in the fact that the M-Process uses system libraries to perform several tasks. The most common of these is the pipe interface of `glibc`. Recall that the attacker has full control of the operating environment. Which means, in Linux, they have full access to the source code. To combat such attacks, it is recommended that proprietary interfaces be written for system library functionality. Pipes and random number generation are the most vulnerable to such attacks.

Not only are system libraries a security threat, they also pose performance problems. When an application makes a system call, there is significant overhead that the kernel must handle. First, the call is now in system space, invoking the kernel for assistance. This means that the application must wait for the kernel thread to execute, in order to perform the operation. In the case of reading or writing to a pipe, the kernel must also keep track of which pipes are open, which process is allowed to read/write to a pipe, and memory management.

Initially it was found that sending the unique identifiers to the pipes at every basic block was significantly slow. To help reduce the performance loss due to system calls, an internal buffer is used. This buffer is used to hold a number of identifiers. Once it has queued up to its limit, it will send them all in one chunk to the M-Process. In much the same fashion, the M-Process will request as many identifiers as its buffer size. The `read()` system call under Linux, is implemented as a "blocking read", meaning that when a process calls the `read()` function, it

will be put to sleep until the number of bytes that it requested have arrived.

3.4.3 Dynamic Identifiers and the CFG Template

A dynamic identifier is a basic block identifier that changes with each run of the program. The driving force behind a dynamic approach is to thwart static analysis of the program. In order to *break* the tamper resistant measures in place, one would only need the CFG for the program. Once they have the CFG, any changes to the flow of the code can be patched up appropriately with the correct values from the CFG. This causes much concern, because of the static nature of the CFG. In other words, the adversary needs to only figure this out once, and it will remain true for the entire lifetime of the application (or at least one version of it).

In order to take this into account, the CFG that is calculated at compile time is used merely as a template. This template will serve as the road map for the identifiers that are placed at runtime. For example, take the identifier *ATP_270* from Figure 3.3. The CFG that was determined at compile time will have references to identifier 270. One tuple will have 270 as the predecessor, and any number of tuples may contain 270 as a successor of another block.

At runtime, the M-Process will access each basic block in the P-Process and update the instruction sequence accordingly to insert the new identifier. Then the template is used to map the new unique identifiers creating a new CFG inside the M-Process, which holds the actual values in use. Use of a static algorithm is not recommended to create the unique identifiers at runtime. Using an algorithm to change the unique identifiers is also static in nature, and easily attackable. For our purposes, the use of the system library function *rand()* was employed.

3.4.4 M and P Process Coupling

A unique complementary M-Process is created for each P-Process in much the same way pieces of a puzzle fit together. In order to execute any P-Process, there is only one M-Process that has the required information to perform the task. Also, the P-Process is stripped down to a state that is not runnable on any architecture or tool. Several key pieces are missing, and it is the job of the M-Process to patch these missing pieces.

After the executable is created, several pieces are taken out, and inserted as data structures in the M-Process. This allows these key pieces to be guarded by the encryption methods used to guard the M-Process itself.

First, the application's main function is stripped out. This puts the application in a state that is not runnable. The reason for this is that an adversary might be able to perform some analysis on the P-Process, if it is possible to get it to a runnable state which does not use the M-Process. This guarantees that running the P-Process by itself will cause a fatal error and it will be discarded by the operating system.

When the P-Process is compiled, it is linked against a common set of code that implements the communication functionality with the M-Process. It is feasible that an attack may *sterilize* the communication by modifying these functions. In order to combat such a scenario, these functions are also stripped out, and placed into the M-Process.

Another sanity check that the M-Process will do is to verify the file size of the P-Process. After the P-Process is compiled, the size of the final executable is known. This value is placed into the M-Process for later use. When the M-Process opens up the P-Process file, the first thing it does is to verify that the sizes match. This means that an adversary cannot insert extra code into the file in the form of attacks. To further enhance this mechanism, a checksum could also be calculated on the P-Process. Some care must be taken when computing the checksum, due to the fact that the P-Process goes through several transformations.

3.4.5 Data Hiding

When the P-Process is started by the M-Process, it needs to know how to communicate its unique identifiers. Namely, it needs the pipe identifier that the M-Process will be listening to. In order to facilitate this process, a placeholder is present in the P-Process' data section. Initially it contains some bogus value, which is of little concern. What is of concern is the flag variable in the P-Process data section that tells the P-Process that the pipe identifier has been updated to the current runtime value. There are several other flags of this nature that the M-Process

will change before the P-Process starts.[3]

A possible attack scenario would involve editing the default values of some or all of these flags, consequently making the P-Process think that the M-Process has correctly patched it, and thus not catching the fact that it has been modified. Such an attack alone would not do much, but it could be a significant building block, when combined with other attack methods.

Data Section Hiding is the process of stripping out the P-Process data section, and placing it inside the M-Process as a data structure. Also remember that the M-Process is protected using encryption techniques. This means that the real data section is only visible for a very short period of time, and depending on the size, may only be partially visible at any given time as well. When the M-Process is in the P-Process fix up phase, it will restore the data section and also modify any values or flags that the P-Process needs for communication.

3.4.6 Instruction Hiding

A common tamper mechanism is to modify key pieces of executable code to *fake out* the application. One such attack would be to nullify the communication with the M-Process. Because of performance reasons, the communication is one-way. The P-Process only sends data, and the M-Process only processes the data it receives. To exploit this fact, one could simply place a return instruction as the first instruction in the function which performs the communication to the M-Process. The function would simply return, and the P-Process would continue running (as would the M-Process). To subvert such an attack, the ability to strip out executable code, and place it as data in the M-Process has been added. When the M-Process starts up, it must replace several key pieces of the P-Process such as pipe communication routines and the *main* routine.

3.4.7 Benefits of a Hardware Implementation

The key hurdle in an actual implementation is the overall performance of the tamper resistant code. A purely software solution has shown itself to be rather inefficient, thus slowing things

[3]The M-Process uses the Binary File Descriptor (BFD) library to edit the P-Process.

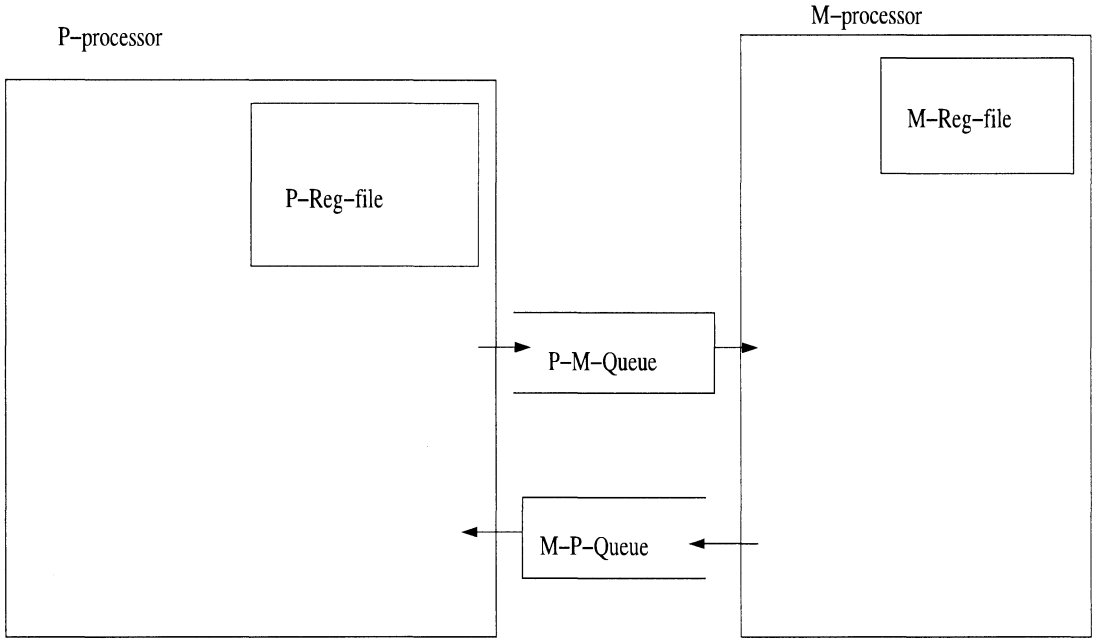

Figure 3.9 Microarchitecture to support M-Process

down. While several enhancements have been made, such as caching, and reduction of system calls, the overall performance can be vastly improved if a hardware based approach is used. The details are beyond the scope of this document, but one can easily see that a hardware M-Process would have its advantages. The verification engine's performance would increase significantly, and the communication could be done with new instructions, also speeding things up.

There are several advantages from a security standpoint as well. The inner workings of the M-Process could be further hidden "inside the chip". This would protect against buss snooping, and other hardware monitoring types of attacks. Note that the two processes P and M are fairly tightly coupled. Hence if a two instruction stream processor microarchitecture to allow for the synchronization between the two streams is available, the overhead of the P and M process interaction will go down significantly. Large part of the overhead is in the operating system based signaling. All of that would be replaced by hardwired signaling, which would be significantly cheaper. Such a processor microarchitecture for a branch decoupled processor [10] has been proposed and evaluated. Figure 3.9 shows the elements of such a microarchitecture.

The salient parts of this architecture are the synchronization queues P-to-M-Queue and M-to-P-Queue. These queues can be destinations of any instructions to deposit either a synchronization token into this queue, and/or to share the value of a register. The two logical processors, P and M, have their own register files. They can also have their own instruction caches.

CHAPTER 4. PERFORMANCE RESULTS

One of the main goals of our approach was to make it appear, from a user's perspective, as if nothing was any different. Because of the way we designed our architecture, it allowed the inner workings of our scheme to be transparent to the user, so the actual performance degradation observable to the user became a final hurdle. We found that the system call overhead was of much concern because of the large number of writes performed to the communication pipe. We also found that, when the CFG table gets to be significantly large, searching this can become time consuming as well.

In order to account for these performance weaknesses, an identifier buffer and cache were implemented. The identifier buffer allows identifiers to be queued and sent all at once, incurring one system call, instead of and entire buffer's worth of system calls. The identifier cache allows the verification engine to quickly find recently used identifiers. Because of the locality of reference of a large number of instructions, a cache is an effective way to avoid searching the entire CFG table. To further speed up CFG searching, a binary search was added. It was also found that because the cache search is a linear search, only a small cache is needed when used in conjunction with the CFG binary search. The following performance results will show how different enhancements to the overall architecture achieve significant performance gains.

The following data shows performance results obtained with the Linux *time* function[1]. The time function takes a program as its argument, and measures the program's execution time. It is only an approximation due to the inaccurate modeling of the system mechanism (interrupts) to schedule the measurement events. However, even these imprecise measurements suffice from a comparative point of view. The results are parametrized by the pipe buffer size, and the

[1]Test system: 2.4Ghz Athlon, 512MB RAM, 266MHz system buss

cache size. The x-axis shows the overall time taken to complete the tests. The y-axis shows the parameter that was varied for each test.

Figure 4.1 shows the execution time results for the well known compression utility, `gzip`. `gzip-1.2.4` in particular was used. Figure 4.2 shows the results for a network simulator, which simulates a prioritized version of the MPCP protocol. The simulation length was set at 0.10 seconds.

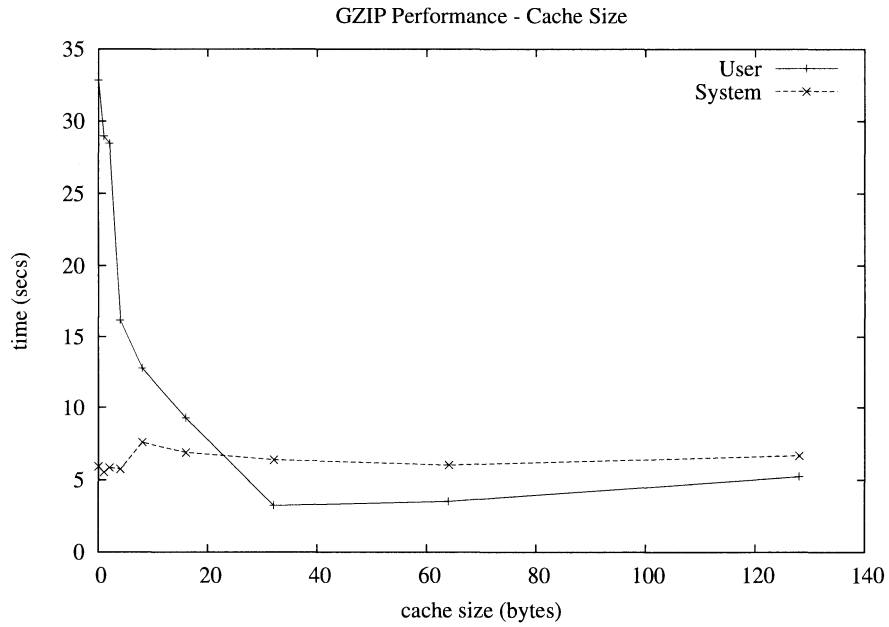

Figure 4.1 Cache results using gzip on a 450kB file.

As is evident, the performance without the buffered control trace verification and CFG caching enhancements is poor. The system time was significantly decreased by lowering the number of system calls (which were mostly needed for the pipe communication). A custom communication method, if correctly designed, could yield even better performance. The user execution time was decreased by implementing an identifier cache. The performance is improved even further by using a binary search with a sorted CFG table.

These performance measures are based on applications which are computationally intensive programs such as gzip. For every OS allocated CPU timeslice, these programs use almost all of

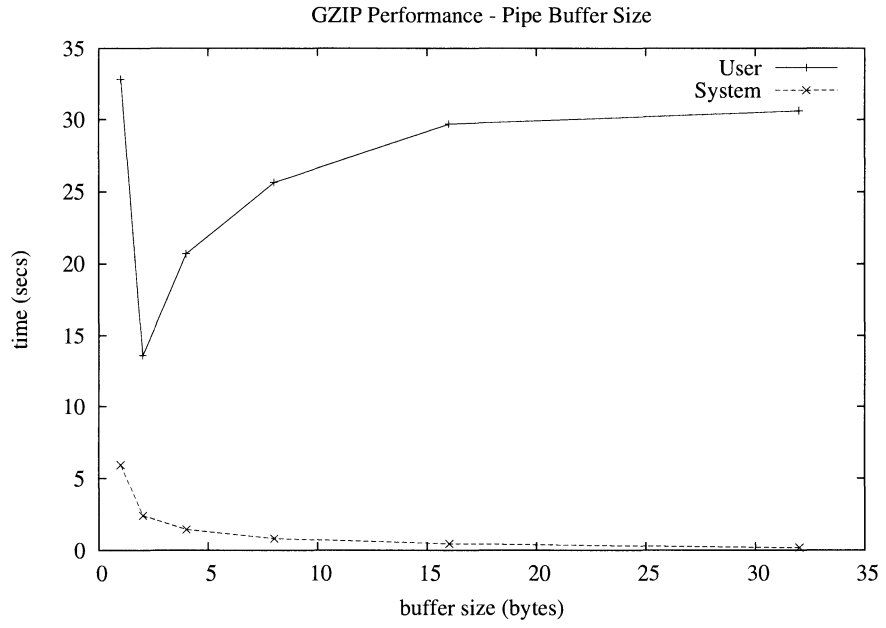

Figure 4.2 Buffer size results using gzip on a 450kB file.

it. In applications that have inherent *sleep* times, such as waiting for user I/O, semaphores, or network I/O, performance degradation is minimal.

For example, Figure 4 shows a clone of a well known game, Breakout. It was used to test performance on the general class of user interaction programs. Figure 4 shows a screenshot of the game. This game is a good test of performance because of not only the user interaction, but also due to the fact that it needs CPU time consistently to display the constantly moving ball. Results showed no jitter in either the ball movement, or the paddle movement as a result of dynamic monitoring of the control flow. For such programs, the quality of the user interaction is the ultimate test of the acceptable performance overhead. It appears as if the proposed anti-tamper technique is more than acceptable for such programs.

Similarly Figure 4 shows a simple calculator written for the Linux Gnome desktop environment called `Galculator`, which uses the GTK graphics libraries. This type of application involves relatively little user interaction. Most of the time, the application is sleeping, waiting for input from the user. There was absolutely no user distinguishable difference in the performance

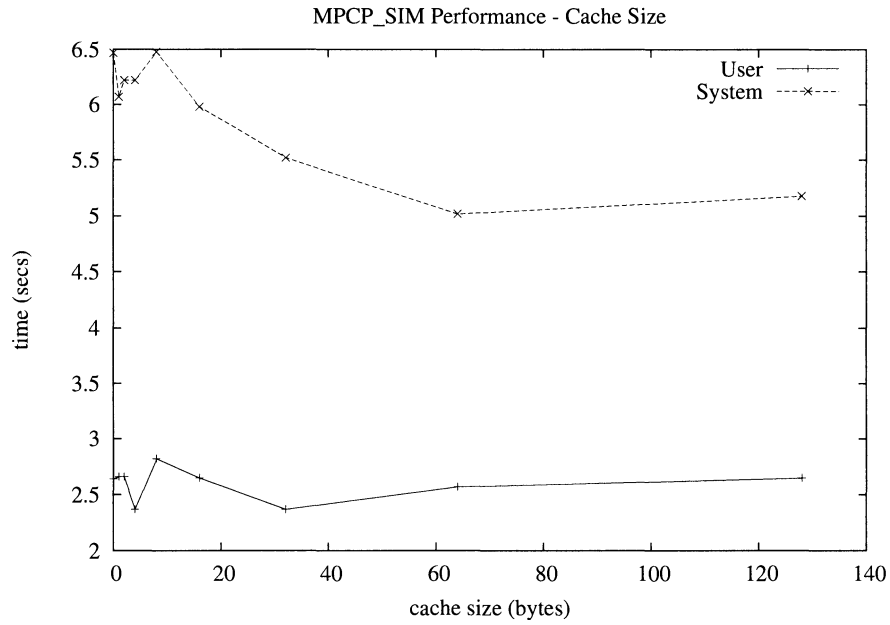

Figure 4.3 Cache results using mpcp_sim for a .1 second simulation.

of the anti-tamper version of Galculator when compared to the original one.

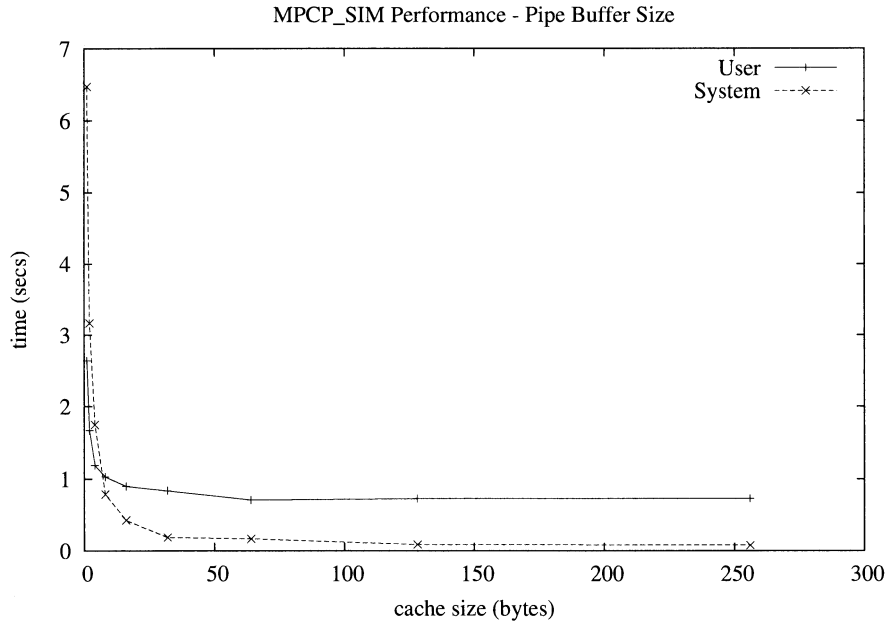

Figure 4.4 Buffer size results using mpcp_sim for a .1 second simulation.

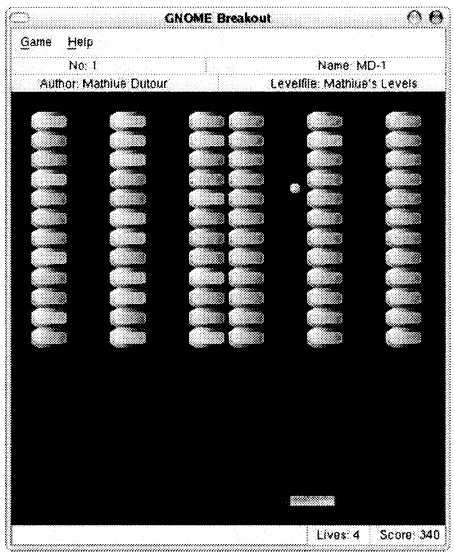

Figure 4.5 Breakout screenshot.

Figure 4.6 Galculator screenshot.

CHAPTER 5. CONCLUSIONS

We have proposed a novel two-process based anti-tamper scheme, wherein a monitoring process monitors the control flow integrity of the monitored process constantly. We implemented such a scheme with gcc which compiles into two such co-processes: one for monitoring and one for the original monitored program. We verified empirically that the performance overhead for user interaction dominated programs is not even observable. For the CPU dominated programs, the implementation can be performed at a variety of axes to trade anti-tamper degree with efficiency. We also propose a two-instruction stream processor microarchitecture to perform the same task with much higher efficiency and more stealth.

Areas such as system libraries, hardware modules and verification triggering deserve further investigation. System libraries should be replaced with an anti-tamper version of the library. This would allow trace information to extend into library calls, thus protecting against library replacement, and/or tampering. Hardware modules would allow for further protection of the CFG table, and the M-Process. Verification is currently triggered at the end of a function, which creates a gap in which to insert tampering code. This is currently the obvious place to "hijack" the P-Process. Although, the task of doing so would require a significant effort due to file size checking and the fact that an adversary must find a safe place to return control back to the P-Process.

BIBLIOGRAPHY

[1] David Aucsmith. "Tamper Resistant Software: An Implementation". *Proceedings of the First International Workshop on Information Hiding*, 1996.

[2] Christian Collberg and Clack Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection, 2000.

[3] Jun Ge. "Software Obfuscation with Program Permutation". *M.S. Thesis*, Dept. of Computer Science, Iowa State University, Ames, IA, in preparation, 2004.

[4] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, M. Violante. "Soft-error Detection Using Control Flow Assertions". *Politecnico di Torino, Dipartimento di Automatica e Informatica*, Torino, Italy 2003

[5] B. Horne, L. Matheson, C. Sheehan, and R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *ACMWorkshop on Security and Privacy in Digital Rights Management*, 2002.

[6] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software, 2000.

[7] Cullen Linn, Saumya Debray, and John Kececioglu, Enhancing Software Tamper-Resistance via Stealthy Address Computations. In *Proceedings of 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Decemeber 2003.

[8] Brian W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[9] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997.

[10] Akhilesh Tyagi. Branch Decoupled Architectures. In *Proc. of Workshop on Interaction between Compilers and Computer Architectures at 3rd Int'l Symp. on High-Performance Computer Architecture*, Feb 1997.

[11] Ramarathnam Venkatesan, Vijay V. Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Information Hiding*, pages 157–168, 2001.

[12] C. Wang, J. Hill, J. Knight, and J. Davidson. "Software Tamper Resistance: Obstructing Static Analysis of Programs". *Technical Report CS2000-12*, Department of Computer Science, University of Virginia, 2000.